Cognome e Nome: _____ **Matricola:** _____ **Mod 2:** ☐ **Mod 1+2:** ☐

Architettura degli Elaboratori (modulo II)

(Compito 1 Giugno 2021)

Esercizio 1

Si consideri il seguente codice MIPS per un'architettura pipeline con *delayed branch*:

```
1.    lw   $5, 0($6)
2.    sw   $8, 0($5)
3.    add  $8, $5, $7
4.    or   $6, $4, $4
5.    bne  $5, $0, label
6.    nop
```

1. Mostrare le dipendenze RAW.
2. Data l'architettura a pipeline a 5 stadi vista a lezione con hazard control unit, forwarding, register file speciale, disegnare il diagramma temporale di esecuzione. Considerare inoltre che il branch che termina la sua esecuzione allo stadio ID per ridurre gli stalli.
3. Discutere quali dipendenze vengono risolte impiegando il forwarding (indicando se il dato viene prelevato dal reg EX/ME o ME/WB) o il register file speciale.
4. E' possibile modificare il codice in modo da ridurre gli stalli e/o riempire il branch delay slot con un'istruzione utile?

Soluzione

1. Le dipendenze sono $1 \rightarrow 2$ (reg. \$5), $1 \rightarrow 3$ (reg. \$5), $1 \rightarrow 5$ (reg. \$5).
2. Il diagramma di esecuzione è il seguente:

```
              1   2   3   4   5   6   7   8   9   10  11
1. lw        IF  ID  EX  ME  WB
2. sw            IF  ID <ID> EX  ME  WB
3. add               IF <IF> ID  EX  ME  WB
4. or                        IF  ID  EX  ME  WB
5. bne                           IF  ID  EX  ME  WB
6. nop                               IF  ID  EX  ME  WB
```

3. La dipendenza $1 \rightarrow 2$ viene risolta tramite forwarding (tra reg. ME/WB e ingresso stadio EXE).
   La dipendenza $1 \rightarrow 3$ viene risolta tramite registrer file speciale.
   La dipendenza $1 \rightarrow 5$ viene risolta naturalmente (WB di 1 avviene prima di ID di 5).
4. L'istruzione `or` è indipendente. L'altra istruzione che può essere spostata solo in avanti è la `add`, perché esiste una dipendenza WAR con la precedente `sw`. Il codice ottimizzato, che elimina tutti gi stalli e riempe il brach delay slot, è il seguente:

```
1.    lw   $5, 0($6)
4.    or   $6, $4, $4
2.    sw   $8, 0($5)
5.    bne  $5, $0, label
3.    add  $8, $5, $7
```

   Il nuovo diagramma di esecuzione è il seguente:

```
              1   2   3   4   5   6   7   8   9
1. lw        IF  ID  EX  ME  WB
4. or            IF  ID  EX  ME  WB
2. sw                IF  ID  EX  ME  WB
5. bne                   IF  ID  EX  ME  WB
3. add                       IF  ID  EX  ME  WB
```

5. La dipendenza $1 \rightarrow 2$ viene risolta tramite forwarding (tra reg. ME/WB e ingresso stadio EXE).
   La dipendenza $1 \rightarrow 3$ viene risolta naturalmente (ID viene eseguito al ciclo 6, quando la WB ha completato).
   La dipendenza $1 \rightarrow 5$ viene risolta grazie al register file speciale.

Esercizio 2

Considerare un sistema di memoria virtuale paginata, dove la **TLB** è associativa a 2 vie, ha un numero totale di 32 ingressi, mentre la TAG della TLB ha dimensione 16 b.
Inoltre, la dimensione della pagina è 4 KB, mentre l'indirizzo fisico ha dimensione 30 b.
La gerarchia di memoria si completa con una **cache associativa** a 4 vie. Inoltre, l'INDEX è di 8 b, mentre la dimensione del blocco è di 16 B.

Rispondere alle seguenti domande:

1. Determinare la dimensione dell'indirizzo virtuale.
2. Determinare la TAG della cache.

Soluzione

1. Gli insiemi della TLB sono $\frac{32}{2} = 16$, per cui INDEX $= \log_2 16 = 4$ b. Inoltre, PAGE_OFFSET $= \log_2 4K = 12$ b. Quindi IND_VIRTUALE = TAG + INDEX + PAGE_OFFSET $= 16 + 4 + 12 = 32$ b.
2. OFFSET $= \log_2$ dim_blocco $= \log_2 2^4 = 4$ b.
   TAG = IND_FISICO - INDEX - OFFSET = 30 - 8 - 4 = 18 b.

Esercizio 3

Tradurre in assembly MIPS la funzione C:
`sum_pos_neg(int *v, int n, int *sum_pos, int *sum_neg)`
i cui risultati dovranno essere memorizzati nelle variabili `sum_pos` e `sum_neg` passate per indirizzo. Per la variabile locale `i` si usi `$s0`, per le altre (`sump` e `sumn`) dei registri temporanei.

```
void sum_pos_neg(int *v, int n, int *sum_pos, int *sum_neg) {
    int i;
    int sump, sumn;

    if (n == 0) return;

    sump = 0;
    sumn = 0;

    for (i=0; i < n; i++) {
        if (*v > 0)
            sump += *v;
        if (*v < 0)
            sumn += *v;
        v = v+1;
     }

    *sum_pos = sump;
    *sum_neg = sumn;
}
```

1. Tradurre in C con `if-goto` e `goto`.
2. Tradurre in assembly MIPS (*E' possibile usare pseudo-istruzioni come `blt`, `ble`, `bgt` e `bge`.*).

Soluzione

Il codice con `if-goto`:

```
void sum_pos_neg(int *v, int n, int *sum_pos, int *sum_neg) {
    int i;  // $s0
    int sump, sumn; // $t1, $t2

    if (n == 0) goto ex_lab;

    sump = 0;
    sumn = 0;

    i = 0;
init_for:
    if (i >= n) goto exit_for;
    if (*v <= 0) goto exit_if1;
    sump += *v;
exit_if1:
    if (*v >= 0) goto exit_if2;
    sumn += *v;
exit_if2:
    v = v+1;
    i++;
    goto init_for;
exit_for:
    *sum_pos = sump;
    *sum_neg = sumn;
ex_lab:
}
```

Traduzione MIPS¿

```
sum_pos_neg:    # $a0: v, $a1: n,  $a2: sum_pos,  $a3: sum_neg
    addiu $sp, $sp, -4
    sw $s0, 0($sp)

    #  int i;  // $s0
    #  int sump, sumn; // $t0, $t1
```

```
    beq $a1, $zero, ex_lab   # if (n == 0) goto ex_lab;

    li $t0, 0                # sump = 0;
    li $t1, 0                # sumn = 0;

    li $s0, 0                # i = 0;
init_for:
    bge $s0, $a1, exit_for   # if (i >= n) goto exit_for;
    lw $t2, 0($a0)            # if (*v <= 0) goto exit_if1;
    ble $t2, $zero, exit_if1
    add $t0, $t0, $t2        # sump += *v;
exit_if1:
    bge $t2, $zero, exit_if2 # if (*v >= 0) goto exit_if2;
    add $t1, $t1, $t2        # sumn += *v;
exit_if2:
    addi $a0, $a0, 4         # v = v+1;
    addi $s0, $s0, 1         # i++;
    j init_for               # goto init_for;
exit_for:
    sw $t0, 0($a2)           # *sum_pos = sump;
    sw $t1, 0($a3)           # *sum_neg = sumn;

ex_lab:
    lw $s0, 0($sp)
    addiu $sp, $sp, 4
    jr $ra
```

Domande

1. Dal punto di vista della banda reale di un dispositivo di I/O, perché conviene usare blocchi grandi, sempre che la l'applicazione specifica lo permetta? Esemplificare.
2. Qual è il compito dell'assemblatore e del linker per generare un file eseguibile? A grandi linee, cosa sono le librerie statiche (SLL) e dinamiche (DLL)?
3. Discutere la località spaziale e temporale. Perché solitamente la località spaziale sulle istruzioni (fetch) è maggiore?
4. Perché la predizione dei salti condizionati è importante, e come viene realizzata nelle CPU moderne?